LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

UCRL-TR-213497

Proposed Babel/SIDL Changes to Support RMI

G. Kumfert, J. Leek

July 11, 2005

This work was performed under the auspices of the U.S. Department of Energy by University of California, Lawrence Livermore National Laboratory under Contract W-7405-Eng-48.

Proposed Babel/SIDL Changes to Support RMI

A Working Document for Babel and Networking Developers
28 October 2004
(updated) 6 December 2004
(updated & released to public) 5 April 2005

UCRL-TR-213497

Gary Kumfert and Jim Leek

Table of Contents

I. Goals....3
II. Current Vision....3
A. Client Side Interactions....3
1. Client Side Creation & Connection....3
2. Client Side Method Invocation....4
3. Client Side Deletion or Disconnect....5
4. Client Side Cast....6
B. Server Side Interactions....7
1. Server Side Creation & Connection....7
2. Server Side Method Invocation....8
3. Server Side Deletion & Disconnect....9
4. Passing of Object References....9
III. Completed and Planned Modifications....10
A. New Builtin Methods....10
1. _create[Remote](in string url)....11
2. _connect(in string url)....11
3. _IHConnect(in InstanceHandle instance)....11
4. _exec(in [selfType] self , in string methName, in sidl_io_Deserializer inArgs, in sidl_io_Serializer outArgs)....11
5. _getURL(in [selfType] self)....12
6. _raddRef(in [selfType] self)....12
7. _isRemote/_isLocal(in [selfType] self)....12
B. Additions to sidl.sidl & sidl runtime library....12
1. sidl.adt.TypedMap class --- (work in progress)....12
2. sidl.rmi.InstanceRegistry class (singleton)....13
3. sidl.io.Serializer interface....13
4. sidl.io.Deserializer interface....14
5. sidl.io.Serializable interface....14
6. sidl.rmi.ProtocolFactory class (singleton)....14
7. sidl.rmi.InstanceHandle interface....15
8. sidl.rmi.Invocation interface....16
9. sidl.rmi.Response interface....16
10. sidl.rmi.ConnectRegistry class (singleton)....17

11. sidl.rmi.ServerInfo interface 17
12. sidl.rmi.ServerRegistry class (singleton).....18
13. New Exception Hierarchy.....18
C. Modifications to Babel Code Generation.....18
1. Changes that affect the user.....19
2. Changes that do not affect the user.....19
D. Non-blocking RMI.....20
1. Non-blocking RMI interface.....20

I. Goals

RMI support in Babel has two main goals: transparency & flexibility. The new RMI features should be transparent to existing Babelized code, allowing painless upgrade. The RMI capability should also be flexible enough to support a variety of RMI transport implementations.

The first goal's ideal would be for Babel users at some future date to simply upgrade to a RMI-enabled Babel release, regenerate files over their existing implementations, and find all their code is now able to be remotable without extensive modifications to their Impl files. The primary strategy for accomplishing the first goal is careful design and implementation of Babel generated code to minimize impact on user code.

The second goal's ideal would be for Babel users to plug in robust WebServices-like modules when accessing Babel objects across a WAN, and utilize faster binary protocol for accessing Babel objects across a LAN, or even different nodes in a leadership-class supercomputer --- without need to recompile their code. The primary strategy for accomplishing this second goal is to partner with appropriate parties to define an RMI API layer (in SIDL) such that various transport mechanisms can be "plugged-in."

II. Current Vision

The current vision for how RMI implementations (which are by nature generic libraries) interact with Babel code (which is generated) is sketched out roughly in this section. Section III will start enumerating foreseeable changes to make this sketch viable. For the purposes of this discussion, we assume the following example:

```
package pkg version 1.0 {
    class cls {
        bool mthd ( in double idbl, out float oflt,
                    in string istr) throws Exception;
  }
}
```

A. Client Side Interactions

First a connection must be established between a stub and a live object elsewhere. We will assume implementations (such as Proteus) may change the actual underlying protocol dynamically. From Babel's point of view, it cares not about the protocol across the wire, only an implementation (SIDL class) that implements the desired SIDL APIs.

1. Client Side Creation & Connection

Occurs when users use either static builtin methods "_create[Remote]()" or " _connect()"

on a Babel object. Both methods create a stub on the local machine that holds a connection to an implementation instance on a remote server. The former creates a new remote instance (and therefore only applies to SIDL classes), of the specific type on the named server using the named RMI implementation. The latter connects to an existing implementation instance on the named server. Connections may occur explicitly, but they will probably more often occur implicitly as objects are passed by reference in argument lists.

Regardless of the language binding, calls to _create[Remote]() will be delegated to the sidl.rmi.ProtocolFactory class implemented in the sidl runtime library[1].

```
sidl_rmi_InstanceHandle ih =
sidl_rmi_ProtocolFactory_createInstance( url, typeName );
```

The ProtocolFactory will in turn parse the URL, identify an implementation associated with the protocol portion of the URL, use sidl.Loader to instantiate it, and initialize it with a call to Connection.init(). The RMI library will implement the sidl.rmi.InstanceHandle interface and Babel stubs when connected to remote objects will store the handle to the sidl.rmi.InstanceHandle IOR in their d_data.

One interesting problem that came up was interface connection. When an interface is passed as an RMI argument, the server must be able to create a stub with an IOR to hold the InstanceHandle. In Babel there is no way to create an interface, only classes may be instantiated, and then cast to interfaces. We solved the problem by generating an anonymous class for each interface that implements only that interface. On connection, this anonymous class in instantiated and cast up to the required interface. The anonymous classname is simply the interface name with a prepended underscore.

2. Client Side Method Invocation

When Stubs to remote objects are created, their EPV is specially initialized for RMI dispatch. The special remote functions that the remote EPV is initialized with must do some special packing and unpacking of arguments. Those Babel generated functions will always be in C, and look like this:

```
sidl_rmi_InstanceHandle = (sidl_rmi_InstanceHandle) self->d_data;
sidl_rmi_Invocation i =
    sidl_rmi_InstanceHandle_createInvocation( c, "mthd");

sidl_rmi_Invocation_packDouble( i, "idbl" , 2.0 );
sidl_rmi_Invocation_packString( i, "istr", "Hello" );
```

1Note: sidl.rmi.InstanceHandle was called sidl.rmi.Connection in the initial draft. Objections were raised that the lifetime of the object potentially spanned multiple TCP/IP connections.

```
sidl_rmi_Response r = sidl_rmi_Invocation_invokeMethod( i );

sidl_rmi_Response_getException(r, _ex);  SIDL_CHECK(*_ex);

sidl_rmi_Response_unpackBool( r, "_retval", &_retval);
sidl_rmi_Response_unpackFloat( r, "oflt", &oflt );
```

Where the RMI libraries implement the following SIDL interfaces: sidl.rmi.InstanceHandle, sidl.rmi.Invocation, and sidl.rmi.Response.

It should also be noted that two different kinds of exceptions may be returned from getException(). Both exceptions thrown from the remote method and network communication exceptions may be returned. invokeMethod() should not throw an exception. All network failures related to the method call should be routed through the Response object. This has the advantage of giving the same route for failure to both blocking and non-blocking RMI. It also simplifies error checking, since there is only one route for failure. The user should still be able to figure out all the important information about the failure from the type of exception thrown.

3. Client Side Deletion or Disconnect

All Babel objects are reference counted and when reference count goes to zero, the object's destructor is invoked. RMI objects are basically the same, but more complex. A Babel remote object has three distinct reference counts. It has a reference count on the stub, a reference count on the Instance Handle, and a remote reference count on the server (see Illustration1). When any of these reference counts reaches zero, the object at that point will be destroyed. The object is completely destroyed when the remote reference count reaches zero.

Babel neither requires nor enforces that the reference count be known exactly at any one location. For instance, the reference count for an object in the IOR could be 2, but one of those could be from Python where Python has 5 internal references to the extension module. In this case, only when all 5 internal references are collected in Python would the IOR's reference count be decremented by one, and only when the IOR's reference count goes to zero would the instance be destroyed.

The main reference count on an object is at the server. Only when that reference count reaches zero is the object actually destroyed. This remote reference count counts the number of InstanceHandles that point to that object.

It is certainly possible that multiple InstanceHandles to a given remote object may exist on the same client. However, since creating and maintaining remote references is expensive, we try to reuse InstanceHandles as freely as possible.

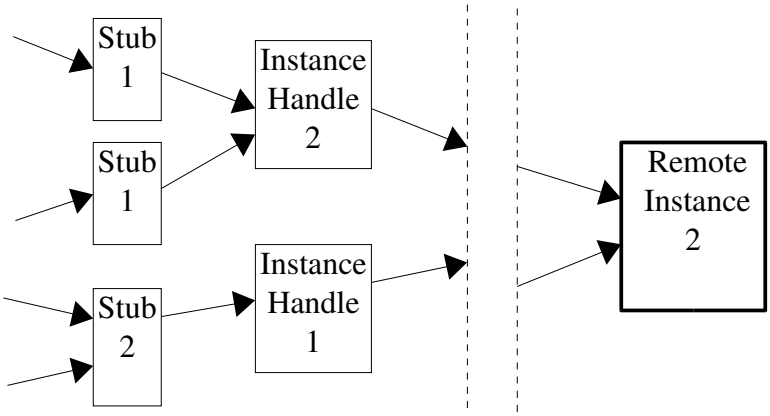

Illustration 1: This shows reference counts on a possible Babel RMI object.

So, in conclusion, The normal add and deleteRef functions will increment and decerement a reference count that only refers to THAT stub. When a stub is collected, it deleteRef's its InstanceHandle, and when an InstanceHandle is collected, it deleteRefs the remote instance. All of this is transparent to the user, who may freely reuse objects in the normal Babel manner.

4. Client Side Cast

With the addition of RMI, casting will now result in incrementing the reference count. In other words, after a cast, the user must deleteRef the pre-cast reference if they do not wish to continue using it.

Traditional Babel casting is fairly simple because the IOR object is always really a complete representation of the most derived type. Casting merely consists of checking if the cast is legal, and, if the cast is to or from an interface, passing back a pointer to a different place in the structure.

In RMI, if an object is passed as an argument, all we know about the object is the location of the object and the declared argument type. We do not know the actual type of the object. We therefore create an stub of that type, and connect it to the remote instance. If, as may often be the case, the argument type is not the most derived type of that object, we will have created a local stub that does not truly represent the “real” type of the remote instance. Therefore, a later RMI object downcast may need to create a new client stub that uses the same InstanceHandle. This is why we must addRef on cast. To understand this, consider the case where a stub, with one reference, is downcasted and creates a new stub. In order to maintain the old babel standard behavior, we would have to deleteRef the old stub, which would destroy it! The user would probably not expect that behavior. This change will have to cascade throughout babel, changing our traditional semantics.

B. Server Side Interactions

Whether the object pre-exists and is posted, or is created on demand by some kind of server, eventually some general library that handles this needs to interact with Babel generated code. The server side interactions are not as strongly scripted as the Client side, because they effect the user much less. However, this document should be sufficent to allow someone to write an ORB for Babel.

A Babel ORB will have 2 basic functions: creation of objects, and allowing function calls on existing objects. Babel already has a mechanism to create instances based on typeName via the sidl.Loader. Babel RMI adds the ability to both (1) access instances based on a string objectID and (2) to execute methods on instances by methodName.

1. Server Side Creation & Connection

We have added the sidl.rmi.InstanceRegistry singleton class with static synchronized methods for mapping objects to objectIDs. All instances that are available remotely are added to the registry. When the last stub deletes its reference, the instance is removed from the registry and destroyed. Note that there is no way for the Babel internals to differentiate between a remote and a local reference. so the object will persist in the registry until *all* references are deleted.

There are three different ways an object may added to the InstanceRegistry, and therefore be published over RMI.

1. A remote machine may call create[Remote] on the currrent host. In this case the object is created and automatically added to the InstanceRegistry as part of it's construction phase. This is a pure remote object, the current host's user code knows nothing of the object, and will only learn of it if the object is passed to the local host as a function argument. The object begins with a reference count of 1, which is held by the remote host. If that remote host calls deleteRef, the object is destroyed.
2. A local object may be implicity added to the InstanceRegistry if it is passed as a function argument to a remote host. Thus the object becomes accessible through RMI, and the remote host gets a reference to the object. The object will remain accessible until its reference count goes to zero and the object is destroyed.
3. A local object may also be explicitly added to the InstanceRegistry by the user. The easiest way would be to simply call getURL on the local object, which will automatically add the object to the InstanceRegistry if it is not already there and return the object's URL. Alternatively, the user could simply add it to the InstanceRegistry manually, but this could potentially cause duplicates in the InstanceRegistry. In either case, the user should be sure to continue to hold a reference to the object as long as they wish to have it published, deleteRef'ing the object could inadvertently destroy it.

2. Server Side Method Invocation

It is assumed that when a remote method is invoked, the RMI library will receive the self-describing stream of data that directs it to dispatch to Babel-generated code. Therefore, the Babel objects that the RMI library interfaces with are defined in SIDL to permit RMI libraries written in multiple languages. The RMI library may multiplex streams to instances on its own, or rely on the InstanceRegistry to get unique handles to instances. The RMI library on the server-side will then perform the following sequence:

```
sidl_io_Deserializer inArgs = ... ;
sidl_io_Serializer outArgs = ...;

sidl_BaseClass bc =
      sidl_InstanceRegistry_getInstance ( "instanceID" );
sidl_BaseClass__exec( bc, "mthd", inArgs, outArgs );
```

And then Babel's "_exec()" method will dispatch to generated code specific to the named method and instance.

```
/* in pkg_cls_mthd__rserver(...) */
double idbl;
float oflt;
char * istr;
SIDL_bool _retval;
SIDL_BaseClass _ex;

sidl_io_Deserializer_unpackDouble( inArgs, "idbl", &idbl );
sidl_io_Deserializer_unpackString( inArgs, "istr", &istr );

_retval = pkg_cls_mthd( self, idbl, oflt, istr, &_ex);

if ( _ex != NULL ) {
    /* register exception and return a handle to it */
} else {
    sidl_io_Serializer_packBool( outArgs, "_retval", _retval );
    sidl_io_Serializer_packFloat( outArgs, "oflt", oflt );
}
```

For cases where argument ordering is all that is required, the string names may be ignored by the RMI library. For cases where the stream is self-describing and arguments may arrive out of order, the RMI library implementor will have to queue up the data for later query. The RMI library will have to implement the sidl.io.Serializer and sidl.io.Deserializer interfaces.

Also, please note that the name of the return value is always “_retval.”

3. Server Side Deletion & Disconnect

Objects are deleted and removed from the InstanceRegistry when their reference count goes to zero. This means that if a user wishes to continue to publish an object over RMI they should continue to hold a reference to an object as long as they wish to publish it.

Users also may explicitly remove an object from the InstanceRegistry by using the remove functions. This however, is highly discouraged. The user has no way of knowing what remote users may be accessing that object. Explicitly removing objects from the InstanceRegistry could have sever unintended consequences.

[Q: How to properly reclaim instances when outstanding references exist in the in the InstanceRegistry but the remote references have gone stale?]

4. Passing of Object References

Great care is taken by the Babel backend to maintain a correct reference count on objects passed as references. There is a set of rules for how the user is expected to addRef and deleteRef Babel objects. Here we will describe the system we used to maintain a correct reference count, despite the added complications of remote instances and the additional reference counting that goes on in remote objects. This may not be an optimal solution, but it is simple and it works. Each argument mode (in, out, inout) has unique problems that must be dealt with when passing objects. We will cover each of these modes and why we chose the solutions we did.

i. In Object Arguments

When objects are passed as in arguments, the callee is not passed a unique reference of the object. If the callee wishes to keep a reference to the object, it must addRef it, thereby declaring that it has its own reference to the object. The callee is NOT allowed to deleteRef the passed in reference. Normally, a reference will not be kept by the callee, and the object argument will retain the same reference count; it will not be either addRef'd or deleteRef'd.

In order to maintain this rule, when a remote reference is passed through Babel, the reference count on the remote server is automatically addRef'd on connect. This creates an RMI object with a stub reference count of one, and instance handle reference count of one, and an incremented remote reference count. When the callee function is exited, the in object reference is deleteRef'd. If the callee did not keep a reference, (ie the library code did not addRef the in object argument) the last deleteRef destroys the stub and instanceHandle, and decrements the remote reference count; thereby maintaining the same reference count in the remote object. If the callee does keep a reference, it will addRef and the local stub count will go up to 2. The deleteRef on return will simply

decrement the stub count to 1, the connection is maintained, the remote count will retain it's incremented reference count.

ii. Out Object Arguments and Return Values

When an object is passed back as an out argument or a return value, the callee function is expected to pass back its own reference. In other words, if the callee wishes to save a reference to the object it is returning, it should addRef it.

This creates a unique problem for Babel. Theoretically, the object should not be deleteRef'd because the reference is passed back to the caller. However, if the object being passed back is remote, not deleteRefing will cause the stub and instanceHandle to be leaked! However, if we pass the reference back and deleteref, the object may be destroyed by the deleteRef before the caller has time to addRef the newly received object. Therefore, we need to "transfer the reference." This means that the server side Babel internals should remote addRef the object, pass it back, and then deleteRef the local copy of it. The caller should then connect without addRefing. (The addRef has already happened remotely on the callee.)

iii. InOut Object Arguments

InOut arguments are similar to out arguments, but in both directions. The caller passes in its reference to the object, meaning the caller must addRef if it wishes to keep a reference to the object. The callee is therefore free to deleteRef the object and replace it with a new one. Again, the callee also passes its reference back to the caller, so it must addRef if it wishes to keep a copy.

Inout arguments are similar to out arguments, and so the solution is similar. Babel transfers the reference both ways. The caller backend code remote addrefs and local deleterefs when calling, and the callee does the same on the return.

III. Completed and Planned Modifications

A. New Builtin Methods

Builtin methods start with a leading underscore (which is not expressible in SIDL) and get generated by the backends for each language. They are never virtual functions.

1. _create[Remote](in string url)

Similar to _create(), this is behaves like a static final method. It differs in that the url argument is parsed to determine the RMI library, remote server, and port number to use in creating a new remote instance. Babel itself is only concerned with the first portion of the url. Babel copies the first portion of the url, up to the first ':' or '+', to find the protocol in the protocol factory. The complete, unmodified url is then passed to the protocol itself for further parsing. The url might take the form:

protocol://server:port/

but the specifics of the format are up to the protocol developer. Babel is only concerned with the first portion (the “scheme”).

_createRemote() exists only for classes and, in the case of failure, returns an empty stub and throws an exception indicative of the failure.

2. _connect(in string url)

Instead of creating a new remote instance, this method creates a new connection to an existing remote object. The object's reference count will be incremented for the duration of the connection. A primary difference between _connect and _create is that interfaces can be connected to remote instances.

Like the url in _create[Remote](), Babel is not concerned with the url beyond the protocol section, but unlike _create[Remote](), the objectID of the remote instance must be communicated somehow. The url might take the form:

protocol://server:port/objectID

3. _IHConnect(in InstanceHandle instance)

This connect is for internal Babel use only. This connect is used in remote object downcasts when we cannot reuse the stub, but can reuse the InstanceHandle. For more information, see the section on casting.

4. _exec(in [selfType] self , in string methName, in sidl_io_Deserializer inArgs, in sidl_io_Serializer outArgs)

This method provides a fundamentally new capability for Babel objects. The _exec function allows all the methods in an object to be invoked by name. The only stipulation being, method name must always be in the long form, since overloading is impossible to support. Essentially, for each method a function is defined that unpacks the in arguments from inArgs , executes the normal EPV entry, packs the out arguments into outArgs, and returns. The _exec function consists for a static table of function string name-function

pointer pairs. When called the _exec function looks up the correct function pointer by name in the table, and calls it.

Although static methods are not supported in RMI, in order to more fully support reflexivity, a static exec builtin function has also been added to Babel. It has the following signature (in the C binding):

```
void pkg_cls__sexec(const char* methodName,
        struct sidl_io_Deserializer__object* inArgs,
        struct sidl_io_Serializer__object* outArgs );
```

5. _getURL(in [selfType] self)

This function returns the URL of the object passed in as self. This URL is used for connection. This function is mostly useful for passing an object by reference, but it also automatically adds objects to the InstanceRegistry if they are not already there. It may be useful in a variety of ways for the user.

6. _raddRef(in [selfType] self)

This function is for internal Babel use only, it is not exposed to the user through the stub. This function increments the remote reference count on a remote object. On a local object, it simply call the normal addRef function. This is only used for "transferring the reference" as described in the section on passing object references as arguments.

7. _isRemote/_isLocal(in [selfType] self)

_isRemote is a builtin function that returns true if the object is implemented remotely, false other wise. _isLocal is the logical opposite. This is used inside of Babel to regulate reference counting, it is exposed to the user for convenience.

_isLocal is not actually built into the epv, it is implemented in the stub as logical not _isRemote.

B. Additions to sidl.sidl & sidl runtime library

1. sidl.adt.TypedMap class --- (work in progress)

Almost identical to the decaf.TypeMap implementation. The only problem is that the current implementation is in C++ making extensive use of STL map<>. [Q: should we support nested TypedMaps? CCA spec doesn't and I never understood why]. Would also make sense for the TypedMap to implement Serializer and Deserializer interfaces for ease of implementing RMI libraries for Babel.

2. sidl.rmi.InstanceRegistry class (singleton)

Basically generates unique names for each registered instance, and serves up instances by name.

```
class InstanceRegistry {

  /**
   * register an instance of a class
   *  the registry will return a string guaranteed to be unique
   *  for the lifetime of the process
   */
  static string registerInstance( in sidl.BaseClass instance );

  /**
   * returns a handle to the class based on the unique string
   */
  static sidl.BaseClass getInstance( in string instanceID );

  /**
   * returns a handle to the class based on the unique string
   * and removes the instance from the table
   */
  static sidl.BaseClass removeInstance(in string instanceID);

}
```

3. sidl.io.Serializer interface

Can be implemented by RMI libraries, IO libraries, maybe even checkpoint/restart and process migration. [Note: "pack" is an intrinsic function in F90. The F90 bindings will need to address the problem that some F90 compilers treat intrinsics as reserved words. Does the name have to be pack?]

```
interface Serializer {
  void pack[Bool]( in string key, in bool value ) throws IOException ;
  void pack[Char]( in string key, in char value ) throws IOException ;
  void pack[Int]( in string key, in int value ) throws IOException ;
  void pack[Long]( in string key, in long value ) throws IOException ;
  void pack[Float]( in string key, in float value ) throws IOException ;
  void pack[Double]( in string key, in double value ) throws IOException ;
  void pack[Fcomplex]( in string key, in fcomplex value ) throws IOException ;
  void pack[Dcomplex]( in string key, in dcomplex value ) throws IOException ;
  void pack[String]( in string key, in string value ) throws IOException ;
  void pack[Serializable]( in string key, in Serializable value )
       throws IOException;

  /* similar for packing arrays of values */
  ...
}
```

4. sidl.io.Deserializer interface

Compliments features from Serializer.[Note: "unpack" is an intrinsic function in F90. The F90 bindings will need to address the problem that some F90 compilers treat intrinsics as reserved words.]

```
interface Deserializer {
  void unpack[Bool]( in string key, out bool value ) throws IOException ;
  void unpack[Char]( in string key, out char value ) throws IOException ;
  void unpack[Int]( in string key, out int value ) throws IOException ;
  void unpack[Long]( in string key, out long value ) throws IOException ;
  void unpack[Float]( in string key, out float value ) throws IOException ;
  void unpack[Double]( in string key, out double value ) throws IOException ;
  void unpack[Fcomplex]( in string key, out fcomplex value ) throws IOException ;
  void unpack[Dcomplex]( in string key, out dcomplex value ) throws IOException ;
  void unpack[String]( in string key, out string value ) throws IOException ;
  void unpack[Serializable]( in string key, out Serializable value )
              throws IOException;

  /* similar for unpacking arrays of values */
}
```

5. sidl.io.Serializable interface

This would be implemented by classes that can pack and unpack themselves to a Serializer/Deserializer.

```
interface Serializeable {
  void pack( in Serializer ser );
  void unpack( in Deserializer des );
}
```

6. sidl.rmi.ProtocolFactory class (singleton)

Serves up RMI libraries (e.g. Proteus) using sidl.Loader based on a string that is found in the URL and almost definitely not the same as the class name. Note that all these methods should throw NetworkExceptions.

```
class ProtocolFactory {
  /**
   * Associate a particular prefix in the URL to a typeName
   * <code>sidl.Loader</code> can find.  The actual type is
   *  expected to implement <code>sidl.rmi.InstanceHandle</code>
   * Return true iff the addition is successful (no collisions
   * allowed)
   */
  static bool addProtocol( in string prefix, in string typeName )
      throws NetworkException;

  /**
   * Return the typeName associated with a particular prefix.
```

```
        * Return empty string if the prefix
        */
       static string getProtocol( in string prefix )
           throws NetworkException;

       /**
        * Remove a protocol from the active list.
        */
       static bool deleteProtocol( in string prefix )
         throws NetworkException;

       /**
        * Create a new, initialized connection based on the input
        * string.  Return ni if protocol unknown or Connection.init()
        *  failed.
        */
       static InstanceHandle createInstance( in string url,
                                             in string typeName )
           throws NetworkException;


        /**
         * Create an new connection linked to an already existing
         * object on a remote
         * server.  The server and port number are in the url, the
         * objectID is the unique ID
         * of the remote object in the remote instance registry.
         * Return nil if protocol unknown or InstanceHandle.init()
         * failed.
         */
        static InstanceHandle connectInstance( in string url)
           throws NetworkException;

   }
```

7. sidl.rmi.InstanceHandle interface

Implemented by the RMI library. (Was called "Connection" in earlier versions, but name was changed in acknowledgment that this object may persist over multiple network connections. This interface extends Serializable so that exceptions and remote object references can be exchanged.

```
   interface InstanceHandle extends sidl.io.Serializable {

     /**
      * initialize a connection (intended for use by the
      * ProtocolFactory)
      */
```

```
    bool init[Create](in string url, in string typeName)
        throws NetworkException;

    /**
     * initialize a connection (intended for use by the
     * ProtocolFactory)
     */
    bool init[Connect]( in string url) throws NetworkException;

    /** return the name of the protocol */
    string getProtocol() throws NetworkException;

    /** return the objectID */
    string getObjectID() throws NetworkException;

    /** return the URL*/
    string getURL() throws NetworkException;

    /** create a handle to invoke a named method */
    Invocation createInvocation( in string methodName )
        throws NetworkException;

    /** closes the connection (called be destructor, if not done
     * explicitly) returns true if successful, false otherwise
     * (including subsequent calls)
     */
    bool close() throws NetworkException;
}
```

8. sidl.rmi.Invocation interface

Implemented by the RMI library. Exists for the duration of a single method invocation.

```
interface Invocation extends sidl.io.Serializer {

    /**
     * this method may be called only once at the end of the
     *  object's lifetime
     */
    Response invokeMethod() throws NetworkException;
}
```

9. sidl.rmi.Response interface

Implemented by the RMI library. Encapsulates the response of a single method invocation.

```
interface Response extends sidl.io.Deserializer {
```

```
  /** if returns null, then safe to unpack arguments */
  sidl.BaseException getExceptionThrown() throws NetworkException;

  /** signal that all is complete */
  bool done() throws NetworkException;
}
```

10. sidl.rmi.ConnectRegistry class (singleton)

This class is for Babel internal use only, it maps symbol names to IHConnect functions. This is part of the system used to downcast remote Babel RMI objects.

```
/** Multiple registrations under the same name must protected
*   against in the user code.*/
class ConnectRegistry {
  static void registerConnect( in string key, in opaque func);
  static opaque getConnect( in string key );
  static opaque removeConnect( in string key );
}
```

11. sidl.rmi.ServerInfo interface

This interface must be implemented by the RMI library, probably by the ORB itself. It is solely written to handle the problems that arise in exporting local objects as RMI objects. If a user wishes to have the ability to export local objects over RMI, they must run an ORB and resister that ORB's ServerInfo with the ServerRegistry. (Registration with the ServerRegistry could also be handled by the ORB during construction.)

```
interface ServerInfo {
 /**
  * Creates a url for a local object, finds the protocol,
  * machine, and process information (protocol://server:port)
  */
  string getServerURL(in string objID) throws NetworkException;

 /**
  * Determines if a url points to a local or remote object.
  * Returns the objectID if is local, Null otherwise.
  */
  string isLocal(in string url) throws NetworkException;
}
```

12. sidl.rmi.ServerRegistry class (singleton)

This singleton class is how Babel generally allows access to the current ServerInfo interface. It is recommended that only one ORB be run per Babel process, since the ServerRegistry does not offer support for having multiple ORBs.

```
class ServerRegistry {
  static void registerServer(in sidl.rmi.ServerInfo si);
  static string getServerURL(in string objID);
  static string isLocal(in string url);
}
```

13. New Exception Hierarchy

With RMI comes all the blessings and curses of network communication. Among the curses of network communication is the fact that it is inherently unstable. The developer needs some mechanism to allow recovery in the case of network failure. The best way of doing this is to have any method that uses RMI throw a NetworkException. However, not all network exceptions are equal, some may be recoverable in some cases, and in other cases fatal. Babel needs a more extensive exception hierarchy to communicate specific types of network failure. Some possible network exceptions are:

UnknownHostException - Thrown when Hostname lookup fails. (DNS failure)

BindException - Usually means that the requested port is in use

ConnectException - Connection Refused, Host Down, etc.

NoRouteToHostException - Router is probably down.

TimeOutException - Request timed out.

UnexpectedCloseException - Network dropped connection because of reset,

Software caused connection abort, Connection reset by peer, etc.

ObjectDoesNotExistException - No such object on the server

Protocol Exception - Non specific protocol error

MalformedURLException - The protocol cannot parse the URL

C. Modifications to Babel Code Generation

The addition of RMI is an extensive change to Babel, and therefore requires significant changes to the code generation portion of Babel.

1. Changes that affect the user

Two major changes will effect even a user who does not use RMI:

The first is the possibility of exceptions being thrown from any method. Because one of the goals of Babel RMI is to allow the developer to painlessly use the same code for both interprocess and intraprocess communication, any method may potentially throw a Network Exception. Therefore, as of Babel 0.11.0, all Babel methods will be automatically declared to throw Network Exceptions. Even if the user believes they do not require exceptions, Babel will still generate the extra argument to hold an exception and the code to throw it across the IOR. Furthermore, as long as we are throwing Network Exceptions, we have decided to add a small set of other basic exceptions, such as memory allocation exceptions.

The second is that as of Babel 0.11.0, Babel cast will addRef. This is due the possiblilty that an RMI downcast will create a whole new stub. If we tried to maintain no-addRef cast in RMI, we would have to deleteRef the old stub to maintain the same reference count. If the user downcasted an RMI object with a reference count of one, it could result in the unintentional destruction of the old stub!

There are also two minor changes that we hope will not heavily affect users of Babel. First, cast2 is being removed from the C binding. We need cast to register it's IHConnect function before casting to the target type. This is impossible for cast2 because the cast2 target type is statically unknown. Since very few users use cast2, we hope this will affect very few people. We are also considering the removal of queryInt, which is not used by any users we know of. The default implementation does not really make much sense when coupled with RMI.

2. Changes that do not affect the user

For the in-process case, Babel the stubs are a library's external access point, and the IOR a provides the virtual function table the guarantees correct function dispatch. The stubs for any language binding must be at least partially implemented in C/C++. (If a language is not a C derivative, it must have a two part Stub, one part written in the intended language that exports the Babel function to that language, and one part that calls though the IOR, written in C.) IORs are always implemented in C. For minimal impact to our customers, we intend to generate the extra support code for RMI in the Stub and IOR C files. The new builtin methods listed in Section A will need to be added to each binding on a case-by-case basis.

It is always desirable to share as much code as possible between languages bindings, and

much of the internal RMI code can be shared between languages. Therefore, a new directory to hold the common code generation source was created in:

babel-x.x.x/compiler/gov/llnl/babel/backend/rmi/*.java.

D. Non-blocking RMI

There is significant interest in non-blocking RMI to interleave the computation with communication. Evidence that this technique is critical to achieving performance in SPMD programming is legion. Work on implementing this in RMI is not. Noteworthy is Kate Kehey's graduate work with PARDIS at Indiana University. Unfortunately, her work does not consider the possibility of the exceptions being thrown from a non-blocking RMI.

1. Non-blocking RMI interface

The majority of the responsibility for making non-blocking RMI efficient lies with the RMI library implementors. Luckily, a well implemented non-blocking RMI library will be just as effective for implementing blocking RMI when using the Babel RMI interfaces. In order to make non-blocking RMI effective, Babel will need to define an additional user interface for non-blocking calls. This user interface will probably consist of 2 additional functions for each Babel method, one for marshaling the arguments and calling the function, and one for unmarshaling the arguments from the response.

Expect future revisions of this document to add more design information here.

NOTES: